AMENDMENTS TO THE SPECIFICATION:

Please replace the current title with the following:

SRAM CONTROLLER FOR PARALLEL PROCESSOR ARCHITECTURE INCLUDING A READ QUEUE AND AN ORDER QUEUE FOR HANDLING REQUESTS

Please replace the paragraph beginning line 1 of page 1 with the following rewritten paragraph:

--This application is a continuation of U.S. patent application Serial No. 10/208,264, filed July 30, 2002, now U.S. Patent No. 6,728,845, which is a continuation of U.S. patent application Serial No. 09/387,110, filed August 31, 1999, now U.S. Patent No. 6,427,196.--

Please replace the paragraph beginning at page 2, line 14 with the following rewritten paragraph:

-- The memory controller performs memory reference sorting to minimize delays (bubbles) in a pipeline from an interface to memory. The memory system is designed to be flooded with memory requests that are independent in nature. The memory controller enables memory reference sorting which reduces dead time or a bubble that occurs with accesses to static random access memory (SRAM). With memory references to SRAM, switching current direction on signal lines between reads and writes produces a bubble or a dead time while waiting for current to settle on conductors coupling the SRAM to the SRAM controller. That is, the drivers that drive current on the bus need to settle out prior to changing states. Thus, repetitive cycles of a read followed by a write can degrade peak bandwidth. Memory reference sorting organizes references to memory such that long strings of reads can be followed by long strings of writes. This can be used to minimize dead time in the pipeline to effectively achieve

closer to maximum available bandwidth. Grouping reads and writes improves cycle-time by eliminating dead cycles. The memory controller performs memory reference sorting based on a read memory reference. --

Please replace the paragraph beginning at page 4, line 10 with the following rewritten paragraph:

-- Referring to FIG. 1, a communication system 10 includes a parallel, hardware-based multithreaded processor 12. The hardware-based multithreaded processor 12 is coupled to a bus such as a peripheral component interface (PCI) bus 14, a memory system 16 and a second bus 18. The system 10 is especially useful for tasks that can be broken into parallel subtasks or functions. Specifically hardware-based multithreaded processor 12 is useful for tasks that are bandwidth oriented rather than latency oriented. The hardware-based multithreaded processor 12 has multiple microengines 22a – 22f each with multiple hardware controlled threads that can be simultaneously active and independently work on a task. --

Please replace the paragraph beginning at page 4, line 21 with the following rewritten paragraph:

-- The hardware-based multithreaded processor 12 also includes a central controller or core processor 20 that assists in loading microcode control for other resources of the hardware-based multithreaded processor 12 and performs other general purpose computer type functions such as handling protocols, exceptions, extra support for packet processing where the microengines pass the packets off for more detailed processing such as in boundary conditions. In one embodiment, the central controller or core processor 20 is a Strong Arm® (Arm is a trademark of ARM Limited, United Kingdom) based architecture. The general purpose microprocessor 20 has an operating system. Through the operating system the central controller or core processor 20 can call functions to operate on microengines 22a-22f. The central

controller or core processor 20 can use any supported operating system preferably a real time operating system. For the central controller or core processor implemented as a Strong Arm architecture, operating systems such as, MicrosoftNT® (Microsoft Corporation, Redmond Washington) real-time, VXWorks®(Wind River Systems, Sunnyvale, California) and μ-CUS, a freeware operating system available over the Internet, can be used. --

Please replace the paragraph beginning at page 5, line 7 with the following rewritten paragraph:

-- The hardware-based multithreaded processor 12 also includes a plurality of function microengines 22a-22f. Functional microengines (microengines) 22a-22f each maintain a plurality of program counters in hardware and states associated with the program counters. Effectively, a corresponding plurality of sets of threads can be simultaneously active on each of the microengines 22a-22f while only one is actually [operating] executing at any one time. --

Please replace the paragraph beginning at page 5, line 15 with the following rewritten paragraph:

-- In one embodiment, there are six microengines 22a-22f as shown. Each microengine[s] 22a-22f has capabilities for processing four hardware threads. The six microengines 22a-22f operate with shared resources including memory system 16 and bus interfaces 24 and 28. The memory system 16 includes a [Synchronous Dynamic Random Access Memory] [(] SDRAM [)] controller unit 26a and a [Static Random Access Memory] [(] SRAM [)] controller 26b. SDRAM memory 16a and SDRAM controller 26a are typically used for processing large volumes of data, e.g., processing of network payloads from network packets. The SRAM controller 26b and SRAM memory 16b are used in a networking implementation for low latency, fast access tasks, e.g., accessing look-up tables, memory for the central processor or core processor 20, and so forth. --

Please replace the paragraph beginning at page 6, line 29 with the following rewritten paragraph:

-- The hardware context swapping also synchronizes completion of tasks. For example, two threads could hit the same shared resource e.g., SRAM. Each one of these separate functional units, e.g., the first in first out bus (FBUS) interface 28, the SRAM controller 26a, and the SDRAM controller 26b, when they complete a requested task from one of the microengine thread contexts reports back a flag signaling completion of an operation. When the flag is received by the microengine, the microengine can determine which thread to turn on. --

Please replace the paragraph beginning at page 7, line 7 with the following rewritten paragraph:

-- One example of an application for the hardware-based multithreaded processor 12 is as a network processor. As a network processor, the hardware-based multithreaded processor 12 interfaces to network devices such as a media access controller (MAC)device e.g., a 10/100BaseT Octal MAC controller13a or a Gigabit Ethernet device 13b. In general, as a network processor, the hardware-based multithreaded processor 12 can interface to any type of communication device or interface that receives/sends large amounts of data. Communication system 10 functioning in a networking application could receive a plurality of network packets from the devices 13a, 13b and process those packets in a parallel manner. With the hardware-based multithreaded processor 12, each network packet can be independently processed. --

Please replace the paragraph beginning at page 7, line 20 with the following rewritten paragraph:

-- Another example for use of processor 12 is a print engine for a postscript processor or as a processor for a storage subsystem, i.e., a redundant array of independent disks (RAID) [disk] storage. A further use is as a matching engine. In the securities industry for example, the advent of electronic trading requires the use of electronic matching engines to match orders between buyers and sellers. These and other parallel types of tasks can be accomplished on the system 10. --

Please replace the paragraph beginning at page 7, line 28 with the following rewritten paragraph:

-- The processor 12 includes a bus interface 28 that couples the processor to the second bus 18. Bus interface 28 in one embodiment couples the processor 12 to the so-called FBUS 18 [(FIFO bus)]. The FBUS interface 28 is responsible for controlling and interfacing the processor 12 to the FBUS 18. The FBUS 18 is a 64-bit wide FIFO bus, used to interface to [Media Access Controller (] MAC [)] devices. --

Please replace the paragraph beginning at page 8, line 4 with the following rewritten paragraph:

-- The processor 12 includes a second interface e.g., a PCI bus interface 24 that couples other system components that reside on the PCI 14 bus to the processor 12. The PCI bus interface 24, provides a high speed data path 24a to memory 16 e.g., the SDRAM memory 16a. Through that path data can be moved quickly from the SDRAM 16a through the PCI bus 14, via direct memory access (DMA) transfers. The hardware based multithreaded processor 12 supports image transfers. The hardware based multithreaded processor 12 can employ a plurality of DMA channels so if one target of a DMA transfer is busy, another one of the DMA channels can take over the PCI bus to deliver information to another target to maintain high processor 12 efficiency. Additionally, the PCI bus interface 24 supports target and master operations. Target operations are operations where slave devices on bus 14 access SDRAMs through reads and

writes that are serviced as a slave to target operations. In master operations, the processor core 20 sends data directly to or receives data directly from the PCI interface 24. --

Please replace the paragraph beginning at page 8, line 22 with the following rewritten paragraph:

-- Each of the functional units are coupled to one or more internal buses. As described below, the internal buses are dual, 32 bit buses (i.e., one bus for read and one for write). The hardware-based multithreaded processor 12 also is constructed such that the sum of the bandwidths of the internal buses in the processor 12 exceeds the bandwidth of external buses coupled to the processor 12. The processor 12 includes an internal core processor bus 32, e.g., an Advanced System bus (ASB) [ASB bus (Advanced System Bus)] or AMBA bus that couples the processor core 20 to the memory controller 26a, 26[c]b and to an ASB or AMBA translator 30 described below. The ASB bus is a subset of the so called AMBA bus that is used with the Strong Arm processor core. The processor 12 also includes a private bus 34 that couples the microengine units to SRAM controller 26b, ASB translator 30 and FBUS interface 28. A memory bus 38 couples the memory controller 26a, 26b to the bus interfaces 24 and 28 and memory system 16 including flash read only memory (flashROM) 16c used for boot operations and so forth. --

Please replace the paragraph beginning at page 8, line 8 with the following rewritten paragraph:

-- Referring to FIG. 2, each of the microengines 22a-22f includes an arbiter that examines flags to determine the available threads to be operated upon. Any thread from any of the microengines 22a-22f can access the SDRAM controller 26a, [SDRAM] SRAM controller 26b or FBUS interface 28. The memory controllers 26a and 26b each include a plurality of queues to store outstanding memory reference requests. The queues either maintain order of memory

references or arrange memory references to optimize memory bandwidth. For example, if a thread_0 has no dependencies or relationship to a thread_1, there is no reason that thread 1 and 0 cannot complete their memory references to the SRAM unit out of order. The microengines 22a-22f issue memory reference requests to the memory controllers 26a and 26b. The microengines 22a-22f flood the memory controllers [subsystems] 26a and 26b with enough memory reference operations such that the memory controllers [subsystems] 26a and 26b become the bottleneck for processor 12 operation. --

Please replace the paragraph beginning at page 11, line 10 with the following rewritten paragraph:

-- The FBUS 18 is a standard industry bus and includes a data bus, e.g., 64 bits wide and sideband control for address and read/write control. The FBUS interface 28 provides the ability to input (receive) and output (transmit) large amounts of data using a series of input and output FIFO=s 29a-29b. From the FIFOs 29a-29b, the microengines 22a-22f fetch data from or command the SDRAM controller 26a to move data from a receive FIFO in which data has come from a device on bus 18, into the FBUS interface 28. The data can be sent through memory controller 26a to SDRAM memory 16a, via a direct memory access. Similarly, the microengines can move data from the SDRAM 26a to interface 28, out to FBUS 18, via the FBUS interface 28. --

Please replace the paragraph beginning at page 11, line 21 with the following rewritten paragraph:

-- Data functions are distributed amongst the microengines. Connectivity to the SRAM controller 26a, SDRAM controller 26b and FBUS interface 28 is via command requests. A command request can be a memory request or an FBUS request. For example, a command request can move data from a register located in a microengine 22a to a shared resource, e.g., an SDRAM location, SRAM location, flashROM memory or some MAC address. The commands

are sent out to each of the functional units and the shared resources. However, the shared resources do not need to maintain local buffering of the data. Rather, the shared resources access distributed data located inside of the microengines. This enables microengines 22a-22f, to have local access to data rather than arbitrating for access on a bus and risk contention for the bus. With this feature, there is a 0 cycle stall for waiting for data internal to the microengines 22a-22f. --

Please replace the paragraph beginning at page 12, line 4 with the following rewritten paragraph:

-- The data buses, e.g., ASB bus 30, SRAM bus 34 and SDRAM bus 38 coupling these shared resources, e.g., memory controllers 26a and 26b are of sufficient bandwidth such that there are no internal bottlenecks. Thus, in order to avoid bottlenecks, the processor 12 has a[n] bandwidth requirement where each of the functional units is provided with at least twice the maximum bandwidth of the internal buses. As an example, the SDRAM can run a 64 bit wide bus at 83 MHz. The SRAM data bus could have separate read and write buses, e.g., could be a read bus of 32 bits wide running at 166 MHz and a write bus of 32 bits wide at 166 MHz. That is, in essence, 64 bits running at 166 MHz which is effectively twice the bandwidth of the SDRAM. --

Please replace the paragraph beginning at page 12, line 16 with the following rewritten paragraph:

-- The core processor 20 also can access the shared resources. The core processor 20 has a direct communication to the SDRAM controller 26a to the bus interface 24 and to SRAM controller 26b via bus 32. However, to access the microengines 22a-22f and transfer registers located at any of the microengines 22a-22f, the core processor 20 accesses the microengines 22a-22f via the ASB Translator 30 over bus 34. The ASB translator 30 can physically reside in the FBUS interface 28, but logically is distinct. The ASB Translator 30 performs an address

translation between FBUS microengine transfer register locations and core processor addresses (i.e., ASB bus) so that the core processor 20 can access registers belonging to the microengines 22a-22c. --

Please replace the paragraph beginning at page 13, line 16 with the following rewritten paragraph:

-- Referring to FIG. 3, an exemplary one of the microengines 22a-22f, e.g., microengine 22f is shown. The microengine includes a control store 70 which, in one implementation, includes a random access memory (RAM) of [here] 1,024 words of 32 bits. The RAM stores a microprogram. The microprogram is loadable by the core processor 20. The microengine 22f also includes controller logic 72. The controller logic includes an instruction decoder 73 and program counter (PC) units 72a-72d. The four micro program counters 72a-72d are maintained in hardware. The microengine 22f also includes context event arbiter or switching logic 74. Context event logic 74 receives messages (e.g., SEQ[_]#_EVENT_RESPONSE; FBI_EVENT_RESPONSE; SRAM _EVENT_RESPONSE; SDRAM _EVENT_RESPONSE; and [ASB] AMBA_EVENT_RESPONSE) from each one of the shared resources, e.g., SRAM 26a, SDRAM 26b, or processor core 20, control and status registers, and so forth. These messages provide information on whether a requested function has completed. Based on whether or not a function requested by a thread has completed and signaled completion, the thread needs to wait for that completion signal, and if the thread is enabled to operate, then the thread is placed on an available thread list (not shown). The microengine 22f can have a maximum of e.g., 4 threads available. --

Please replace the paragraph beginning at page 14, line 7 with the following rewritten paragraph:

-- In addition to event signals that are local to an executing thread, the microengines 22 employ signaling states that are global. With signaling states, an executing thread can broadcast a signal state to all microengines 22. [Receive Request Available signal,] Any and all threads in the microengines can branch on these signaling states. These signaling states can be used to determine availability of a resource or whether a resource is due for servicing. --

Please replace the paragraph beginning at page 14, line 15 with the following rewritten paragraph:

-- The context event arbiter or logic 74 has arbitration for the four (4) threads. In one embodiment, the arbitration is a round robin mechanism. Other techniques could be used including priority queuing or weighted fair queuing. The microengine 22f also includes an execution box (EBOX) data path 76 that includes an arithmetic logic unit 76a and general purpose register set 76b. The arithmetic logic unit 76a performs arithmetic and logical functions as well as shift functions. The register[s] set 76b has a relatively large number of general purpose registers. As will be described in FIG. 3B, in this implementation there are 64 general purpose registers in a first bank, Bank A and 64 in a second bank, Bank B. The general purpose registers are windowed as will be described so that they are relatively and absolutely addressable. --

Please replace the paragraph beginning at page 14, line 29 with the following rewritten paragraph:

-- The microengine 22f also includes a write transfer register stack 78 and a read transfer register stack 80. These registers are also windowed so that they are relatively and absolutely addressable. Write transfer register stack 78 is where write data to a resource is located. Similarly, read register stack 80 is for return data from a shared resource. Subsequent to or concurrent with data arrival, an event signal from the respective shared resource e.g., the SRAM

controller 26a, SDRAM controller 26b or core processor 20 will be provided to context event arbiter 74 which will then alert the thread that the data is available or has been sent. Both transfer register banks 78 and 80 are connected to the execution box (EBOX) 76 through a data path. In one implementation, the read transfer register stack 80 has 64 registers and the write transfer register stack 78 has 64 registers. --

Please replace the paragraph beginning at page 15, line 12 with the following rewritten paragraph:

-- As shown in FIG. 3A, the microengine datapath maintains a 5-stage micro-pipeline 82. This pipeline includes lookup of microinstruction words 82a, formation of the register file addresses 82b, read or fetch of operands from register file 82c, ALU, shift or compare operations 82d, and write-back of results to registers 82e. By providing a write-back data bypass into the ALU/shifter units, and by assuming the registers are implemented as a register file (rather than a RAM), the microengine can perform a simultaneous register file read and write, which completely hides the write operation. --

Please replace the paragraph beginning at page 24, line 24 with the following rewritten paragraph:

-- There are at least two general operational paradigms from which microengine or microcontroller micro-programs could be designed. One would be that overall microengine or microcontroller compute throughput and overall memory bandwidth are optimized at the expense of single thread execution latency. This paradigm would make sense when the system has multiple microengines or microcontrollers executing multiple threads per microengine or microcontroller on unrelated data packets. --

Please replace the paragraph beginning at page 24, line 31 with the following rewritten paragraph:

-- A second one is that microcontroller or microengine execution latency should be optimized at the expense of overall microengine compute throughput and overall memory bandwidth. This paradigm could involve execution of a thread with a real-time constraint, that is, a constraint which dictates that some work must absolutely be done by some specified time. Such a constraint requires that optimization of the single thread execution be given priority over other considerations such as memory bandwidth or overall computational throughput. A real-time thread would imply a single microengine that executes only one thread. Multiple threads would not be handled because the goal is to allow the single real-time thread to execute as soon as possible--execution of multiple threads would hinder this ability. --

Please replace the paragraph beginning at page 24, line 13 with the following rewritten paragraph:

-- The coding style of these two paradigms could be significantly different with regard to issuing memory references and context switching. In the real time case, the goal is to issue as many memory references as soon as possible in order to minimize the memory latency incurred by those references. Having issued as many references as early as possible the goal would be to perform as many computations [as] in the microengines as possible in parallel with the references. A computation flow that corresponds to real-time optimization is: --

Please replace the paragraph beginning at page 26, line 15 with the following rewritten paragraph:

-- Referring to FIG. 3C, the two register address spaces that exist are Locally [accessibly] accessible registers, and Globally accessible registers accessible by all microengines. The General Purpose Registers (GPRs) are implemented as two separate banks (A bank and B bank) whose addresses are interleaved on a word-by-word basis such that A bank registers have lsb=0, and B bank registers have lsb=1. Each bank is capable of performing a simultaneous read and write to two different words within its bank. –

Please replace the paragraph beginning at page 26, line 24 with the following rewritten paragraph:

-- Across bank[s A and] B, the register set 76b is also organized into four windows 76b0-76b3 of 32 registers that are relatively addressable per thread. Thus, thread_0 will find its register 0 at 77a (register 0), the thread_1 will find its register_0 at 77b (register 32), thread_2 will find its register_0 at 77c (register 64), and thread_3 at 77d (register 96). A similar scheme applies to Bank A but reference numerals are only denoted for Bank B. Relative addressing is supported so that multiple threads can use the exact same control store and locations but access different windows of register and perform different functions. The uses of register window addressing and bank addressing provide the requisite read bandwidth using only dual ported RAMS in the microengine 22f. --

Please replace the paragraph beginning at page 28, line 8 with the following rewritten paragraph:

-- If <a6:a5>=1,1, <b6:b5>=1,1, or <d7:d6>=1,1 then the lower bits are interpreted as a context-relative address field (described below). When a non-relative A or B source address is specified in the A, B absolute field, only the lower half of the SRAM/ASB and SDRAM address spaces can be addressed[.], [Effectively]effectively, reading absolute SRAM/SDRAM devices

[has the effective address space; however] . However, since this restriction does not apply to the dest field, writing the SRAM/SDRAM still uses the full address space. --

Please replace the paragraph beginning at page 28, line 17 with the following rewritten paragraph:

-- In relative mode, addresses [a] are specified as an address [is] offset within context space as defined by a 5-bit source field (a4-a0 or b4-b0): --

Please replace the paragraph beginning at page 29, line 10 with the following rewritten paragraph:

-- The microengines are not interrupt driven. Each microflow executes until completion and then a new flow is chosen based on the state signaled by other devices in the processor 12. --

Please replace the paragraph beginning at page 29, line 13 with the following rewritten paragraph:

-- Referring to FIG. 4, the SDRAM memory controller 26a includes [memory reference] address and command queues 90 where memory reference requests arrive from the various microengines 22a-22f. The memory controller 26a includes an arbiter 91 that selects the next the microengine reference requests to go to any of the functioning units. Given that one of the microengines is providing a reference request, the reference request will come through the address and command queue 90, inside the SDRAM controller 26a. If the reference request has a bit set called the "optimized MEM bit" the incoming reference request will be sorted into either the even bank queue 90a or the odd bank queue 90b. If the memory reference request does not have a memory optimization bit set, the default will be to go into an order queue 90c. The SDRAM controller 26a is a resource which is shared among the FBUS interface 28, the core

processor 20 and the PCI interface 24. The SDRAM controller 26a also maintains a state machine for performing READ-MODIFY-Write atomic operations. The SDRAM controller 26a also performs byte alignment for requests of data from SDRAM. --

Please replace the paragraph beginning at page 30, line 16 with the following rewritten paragraph:

-- The SDRAM controller 26a also includes core bus interface logic [i.e., ASB bus 92]. The ASB bus interface logic 92 interfaces the core processor 20 to the SDRAM controller 26a. The ASB bus is a bus that includes a 32 bit data path and a 28 bit address path. The data is accessed to and from memory through MEM ASB data device 98, e.g., a buffer. MEM ASB data device 98 is a queue for write data. If there is incoming data from the core processor 20 via ASB interface logic 92, the data can be stored into the MEM ASB device 98 and subsequently removed from MEM ASB device 98 through the SDRAM interface 110 to SDRAM memory 16a. Although not shown, the same queue structure can be provided for the reads. The SDRAM controller 26a also includes an engine 97 to pull data from the microengines and PCI bus. --

Please replace the paragraph beginning at page 30, line 1 with the following rewritten paragraph:

-- The order queue 90c maintains the order of reference requests from the microengines. With a series of odd and even bank[s] references it may be required that a signal is returned only upon completion of a sequence of memory references to both the odd and even banks. If the microengine 22f sorts the memory references into odd bank and even bank references and one of the banks, e.g., the even bank is drained of memory references before the odd bank but the signal is asserted on the last even reference, the memory controller 26a could conceivably signal back to a microengine that the memory request had completed, even though the odd bank reference

had not been serviced. This occurrence could cause a coherency problem. The situation is avoided by providing the order queue 90c allowing a microengine to have multiple memory references outstanding of which only its last memory reference needs to signal a completion. --

Please replace the paragraph beginning at page 30, line 16 with the following rewritten paragraph:

-- The SDRAM controller 26a also includes a high priority queue 90d. In the high priority queue 90d, an incoming memory reference from one of the microengines goes directly to the high priority queue and is operated upon at a higher priority than other memory references in the other queues. All of these queues, the even bank queue 90a, the odd bank queue 90b, the order queue 90c and the high priority queue, are implemented in a single RAM structure that is logically segmented into four different windows, each window having its own head and tail pointer. Since filling and draining operations are only a single input and a single output, they can be placed into the same RAM structure to increase density of RAM structures. An insert queue control 95a and a remove queue arbitration 95b logic control insert and removal of memory references from the queues. --

Please replace the paragraph beginning at page 31, line 710 with the following rewritten paragraph:

-- Additional queues include the PCI address queue 94 and [ASB] AMBA read/write queue 96 that maintain a number of requests. The memory requests are sent to SDRAM interface 110 via multiplexer 106. The multiplexer 106 is controlled by the SDRAM arbiter 91 which detects the fullness of each of the queues and the status of the requests and from that decides priority based on a programmable value stored in a priority service control register 100.--

Please replace the paragraph beginning at page 31, line 18 with the following rewritten paragraph:

-- Once control to the multiplexer 106 selects a memory reference request, the memory reference request[,] is sent to a decoder 108 where it is decoded and an address is generated. The decoded address is sent to the SDRAM interface 110 where it is decomposed into row and column address strobes to access the SDRAM 16a and write or read data over data lines 16a sending data to bus 112. In one implementation, bus 112 is actually two separate buses instead of a single bus. The separate buses would include a read bus coupling the distributed microengines 22a-22f and a write bus coupling the distributed microengines 22a-22f. --

Please replace the paragraph beginning at page 31, line 28 with the following rewritten paragraph:

-- A feature of the SDRAM controller 26a is that when a memory reference is stored in the queues 90, in addition to the optimized MEM bit that can be set, there is a "chaining bit". The chaining bit when set allows for special handling of contiguous memory references. As previously mentioned, the arbiter 12 controls which microengine will be selected to provide memory reference requests over the command[er] bus to queue 90 (FIG. 4). Assertion of the chain bit will control the arbiter to have the arbiter select the functional unit which previously requested that bus because setting of the chain bit indicates that the microengine issued a chain request. --

Please replace the paragraph beginning at page 32, line 8 with the following rewritten paragraph:

-- Contiguous memory references will be received in queue 90 when the chaining bit is set. Those contiguous references will typically be stored in the order queue 90c because the contiguous memory references are multiple memory references from a single thread. In order to provide synchronization, the memory controller 26a need only signal at the end of the chained [memory] SDRAM references when done. However, in an optimized memory chaining, (e..g, when optimized MEM bit and chaining bit are set) the memory references could go into different banks and potentially complete on one of the banks issuing the signal "done" before the other bank was fully drained, thus destroying coherency. Therefore, the chain bit is used by the controller 110 to maintain the memory references from the current queue. --

Please replace the paragraph beginning at page 32, line 21 with the following rewritten paragraph:

-- Referring to FIG. 4A, a flow representation of the arbitration policy in the SDRAM controller 26a is shown. The arbitration policy favors chained microengine memory requests. The process 115 starts by examining for Chained microengine memory reference requests at service stops or step 115a . The process step 115 stays at the chained requests until the chain bit is cleared. The process examines ASB bus requests step 115b followed by PCI bus requests 115c, High Priority Queue Service step 115d, Opposite Bank Requests step115e, Order Queue Requests step 115f, and Same Bank Requests step 115g. Chained request are serviced completely, whereas service[s] steps 115b-115d are serviced in round robin order. Only when service[s] steps 115a-115d are fully drained does the process handle service[s] steps 115e-115g. Chained microengine memory reference requests are when the previous SDRAM memory request has the chain bit set. When the chain bit is set then the arbitration engine simply services the same queue again, until the chain bit is cleared. The ASB bus has [is] higher priority than the PCI bus due to the severe performance penalty imposed on the Strong Arm core when the ASB bus is in wait state. PCI has higher priority than the microengines due to the latency requirements of PCI. However with other buses, the arbitration priority could be different. --

Please replace the paragraph beginning at page 33, line 18 with the following rewritten paragraph:

-- Referring to FIG. 5, the memory controller 26b for the SRAM is shown. The memory controller 26b includes an address and command queue 120. While the memory controller 26a (FIG. 4) has a queue for memory optimization based on odd and even banking, memory controller 26b is optimized based on the type of memory operation, i.e., a read or a write. The address and command queue 120 includes a high priority queue 120a, a read queue 120b which is the predominant memory reference function that an SRAM performs, and an order queue 120c which in general will include all writes to SRAM and reads that are to be non-optimized. Although not shown, the address and command queue 120 could also include a write queue. An insert queue control 150a and a remove queue arbitration 150b logic control insert and removal of memory references from the queues. --

Please replace the paragraph beginning at page 33, line 30 with the following rewritten paragraph:

-- The SRAM controller 26b also includes core bus interface logic [i.e., ASB bus 122]. The ASB bus interface logic 122 interfaces the core processor 20 to the SRAM controller 26b. The ASB bus is a bus that includes a 32 bit data path and a 28 bit address path. The data is accessed to and from memory through MEM ASB data device 128, e.g., a buffer. MEM ASB data device 128 is a queue for write data. If there is incoming data from the core processor 20 via ASB interface logic 122, the data can be stored into the MEM ASB device 128 and subsequently removed from MEM ASB device 128 through SRAM interface 140 to SRAM memory 16b. Although not shown, the same queue structure can be provided for reads. The SRAM controller 26b also includes an engine 127 to pull data from the microengines and PCI bus. --

Please replace the paragraph beginning at page 34, line 12 with the following rewritten paragraph:

-- The memory requests are sent to SRAM interface 140 via multiplexer 126. The multiplexer 126 is controlled by the SRAM arbiter 131 which detects the fullness of each of the queues and the status of the requests and from that decides priority based on a programmable value stored in a priority service control register 130. Once control to the multiplexer 126 selects a memory reference request, the memory reference request[,] is sent to a command decoder and address generator138 where it is decoded and an address is generated. --

Please replace the paragraph beginning at page 34, line 20 with the following rewritten paragraph:

-- The SRAM Unit 26b maintains control of the Memory Mapped off-chip SRAM and Expansion read only memory (ROM). The SRAM controller 26b can address, e.g., 16 MBytes, with, e.g., 8 MBytes mapped for SRAM 16b, and 8 MBytes reserved for special functions including: Boot space via flashROM 16c; and Console port access for MAC devices 13a, 13b and access to associated (RMON) counters. The SRAM is used for local look-up tables and queue management functions. --

Please replace the paragraph beginning at page 35, line 1 with the following rewritten paragraph:

-- The SRAM controller 26b performs memory reference sorting to minimize delays (bubbles) in the pipeline from the SRAM interface 140 to SRAM [memory] 16b. The SRAM controller 26b does memory reference sorting based on the read function. A bubble can either be 1 or 2 cycles depending on the type of memory device employed. --

Please replace the paragraph beginning at page 35, line 7 with the following rewritten paragraph:

-- The SRAM controller 26b includes a lock lookup device 142 which is an eight (8 entry address) content addressable memory (CAM) for look-ups of read locks. Each position includes a valid bit that is examined by subsequent read-lock requests. The address and command queue 120 also includes a Read Lock Fail Queue 120d. The Read Lock Fail Queue 120d is used to hold read memory reference requests that fail because of a lock existing on a portion of memory. That is, one of the microengines issues a memory request that has a read lock request that is processed in address and control queue 120. The memory request will operate on either the order queue 120c or the read queue 120b and will recognize it as a read lock request. The controller 26b will access lock lookup device 142 to determine whether this memory location is already locked. If this memory location is locked from any prior read lock request, then this memory lock request will fail and will be stored in the read lock fail queue 120d. If it is unlocked or if device142 shows no lock on that address, then the address of that memory reference will be used by the SRAM interface 140 to perform a traditional SRAM address read/write request to SRAM [memory] 16b. The command decoder [controller] and address generator 138 will also enter the lock into the lock look up device 142 so that subsequent read lock requests will find the memory location locked. A memory location is unlocked by operation of [the] a microcontrol instruction in a program after the need for the lock has ended. The location is unlocked by clearing the valid bit in the CAM. After an unlock, the read lock fail queue 120d becomes the highest priority queue giving all queued read lock misses[,] a chance to issue a memory lock request. --

Please replace the paragraph beginning at page 36, line 9 with the following rewritten paragraph:

-- Referring to FIG. 6, communication between the microengines 22 and the FBUS interface Logic (FBI) is shown. The FBUS interface 28 in a network application [can] performs header processing of incoming packets from the FBUS 18. A key function which the FBUS interface performs is extraction of packet headers, and a microprogrammable source/destination/protocol hashed lookup in SRAM . If the hash does not successfully resolve, the packet header is promoted to the core processor [28] 20 for more sophisticated processing. --

Please replace the paragraph beginning at page 36, line 18 with the following rewritten paragraph:

-- The FBI 28 contains a Transmit FIFO 182, a Receive FIFO 183, a HASH unit 188 and FBI control and status registers (CSR) 189. These four units communicate with the microengines 22, via a time-multiplexed access to the SRAM bus 38 which is connected to the transfer registers 78, 80 (FIG. 3) in the microengines. That is, all communications to and from the microengines are via the transfer registers 78, 80. The FBUS interface 28 includes a push state machine 200 for pushing data into the transfer registers during the time cycles which the SRAM is NOT using the SRAM data bus (part of bus 38) and a pull state machine 202 for fetching data from the transfer registers in the respective microengine. --

Please replace the paragraph beginning at page 36, line 29 with the following rewritten paragraph:

-- The [Hashing] Hash 188 unit includes a pair of FIFO-'s 188a, 188b. The hash unit determines that the FBI 28 received an FBI_hash request. The hash unit 188 fetches hash keys from the calling microengine 22. After the keys are fetched and hashed, the indices are delivered back to the calling microengine 22. Up to three hashes are performed under a single FBI_hash request. The busses 34 and 38 are each unidirectional: SDRAM_push/pull_data, and

Sbus_push/pull_data. Each of these busses require[s] control signals which will provide read/write controls to the appropriate microengine 22 Transfer registers. --

Please replace the paragraph beginning at page 38, line 1 with the following rewritten paragraph:

-- Any microengine 22 that uses the FBI unsolicited push technique must test the protection flag prior to accessing the FBUS interface/microengine agreed upon transfer registers. If the flag is not asserted, then the transfer registers may be accessed by the microengine. If the flag is Asserted then the context should wait N cycles prior to accessing the registers. Initially, [A priori] this count is determined by the number of transfer registers being pushed, plus a frontend protection window. The basic idea is that the microengine must test this flag then quickly move the data which it wishes to read from the read transfer registers to GPR's in contiguous cycles, so the push engine does not collide with the microengine read. --